An Incomplex Algorithm for Fast Suffix Array Construction

Klaus-Bernd Schürmann* Jens Stoye†

Abstract

Our aim is to provide full text indexing data structures and algorithms for universal usage in text indexing. We present a practical algorithm for suffix array construction. The fundamental algorithm is less complex than other construction algorithms. We achieve very fast construction times for common strings as well as for worst case strings by enhancing our basic algorithms with further techniques.

1 Introduction.

Full text indices are used to process sequences of different kind for diverse applications. The suffix tree is the best known full text index. It has been studied for decades and is used in many algorithmic applications. Nevertheless, in practice, the suffix tree is less used than one would expect. We believe that this lack in practical usage is due to the high space requirements of the data structure, and suffix trees are conceptually simple, yet difficult to implement. Already the construction of suffix trees with linear time methods is non-trivial.

The research on suffix arrays increased since Manber and Myers [17] introduced this data structure as an alternative to suffix trees in the early 1990s. They gave an $O(n \log n)$ time algorithm to directly construct suffix arrays, where n is the length of text to be indexed. The algorithm is based on the doubling technique introduced by Karp *et al.* [11]. The theoretically best algorithms to construct suffix arrays run in $\Theta(n)$ time [7]. But until 2003 all known algorithms reaching this bound took the detour over suffix tree construction and afterwards obtaining the order of suffixes by traversing the suffix tree. The drawback of this approach is the space demand of linear time suffix tree construction algorithms. The most space efficient implementation by Kurtz [15] uses between $8n$ and $14n$ bytes of space in total. Moreover, the linear time suffix tree construction algorithms do not explicitly consider the memory hierarchy, which leads to unfavorable effects on current computer architectures. When the suffix tree grows over a certain size, the ratio of cache misses rises.

In 2003, the problem of direct linear time construction of suffix arrays was solved by three independent groups [10, 13, 14]. Shortly after, Hon *et al.* [8] gave a linear time algorithm that needs $O(n)$ bits of working space.

There has also been much progress in practical suffix array construction. Larsson and Sadakane [16] presented a fast algorithm, called *qsufsort*, running in $O(n \log n)$ worst case time using $8n$ bytes. Like Manber and Myers [17] they utilize the doubling technique of Karp *et al.* [11]. Recently, Kim *et al.* [12] introduced a divide-and-conquer algorithm based on [13] with $O(n \log \log n)$ worst-case time complexity, but with faster practical running times than the mentioned linear time algorithms. Both algorithms [12, 13] use the odd-even scheme introduced by Farach [6] for suffix tree construction.

Other viable algorithms are mainly concerned about space. They are called *lightweight algorithms* due to their small space requirements. Itoh and Tanaka [9] as well as Seward [20] proposed algorithms using only $5n$ bytes. In theory their worst-case time complexity is $\Omega(n^2)$. However, in practice, they are very fast, if the average LCP is small. (By LCP we refer to the length of the longest common prefix of two consecutive suffixes in the suffix array.) More recently, Manzini and Ferragina [18] engineered an algorithm called *deep-shallow* suffix sorting. They combine different methods to sort suffixes depending on LCP lengths and spend quite a bit of work on finding suitable settings to achieve fast construction. The algorithm's space demands are small, and it is applicable for strings with high average LCP.

The most recent lightweight algorithm was developed by Burkhardt and Kärkkäinen [3]. It is called *difference-cover* algorithm. Its worst case running time is $O(n \log n)$ by using $O\left(\frac{n}{\sqrt{\log n}}\right)$ extra space. But for common real-life data it is on average slower than *deep-shallow* suffix sorting.

The above mentioned suffix array construction algorithms meet some of the following requirements for practical suffix array construction:

*AG Genominformatik, Technische Fakultät, Universität Bielefeld, Germany, Klaus-Bernd.Schuermann@CeBiTec.Uni-Bielefeld.de.

†AG Genominformatik, Technische Fakultät, Universität Bielefeld, Germany, stoye@TechFak.Uni-Bielefeld.de.

- Fast construction for common real-life strings (small average LCP) – Larsson and Sadakane [16], Itoh and Tanaka [9], Seward [20], Manzini and Ferragina [18], Kim *et al.* [12].

- Fast construction for degenerate strings (high average LCP) – Larsson and Sadakane [16], Manzini and Ferragina [18], Kim *et al.* [12], Burkhardt and Kärkkäinen [3], and others [10, 13, 14, 17].

- Small space demands – Itoh and Tanaka [9], Seward [20], Manzini and Ferragina [18], Burkhardt and Kärkkäinen [3].

Based on our experience with biological sequence data, we believe that further properties are requested. There are many applications where very long sequences with mainly small LCPs, interrupted by occasional very large LCPs are investigated. In genome comparison, for example, concatenations of similar sequences are indexed to find common subsequences, repeats, and non-repeating regions. Thus, to compare genomes of closely related species, one has to build suffix arrays for strings with highly variable LCPs.

We believe that the characteristics as we observe in a bioinformatics context can also be found in other application areas. This, and the fact that in Burrows-Wheeler text compression the problem of computing the Burrows-Wheeler-Transform [4] by block-sorting the input string is equivalent to suffix array construction, stresses the importance of efficient ubiquitous suffix array construction algorithms.

In Section 2 we give the basic definitions and notations concerning suffix arrays and suffix sorting. Section 3 is devoted to our approach, the *bucket-pointer refinement* (*bpr*) algorithm. Experimental results are presented in Section 4. Section 5 concludes and discusses open questions.

2 Suffix Arrays and Sorting – Definitions and Terminology.

Let Σ be a finite ordered alphabet of size $|\Sigma|$ and $t = t_1 t_2 \dots t_n \in \Sigma^n$ a text over Σ of length n. Let \$ be a character not contained in Σ, and assume $\$ < c$ for all $c \in \Sigma$. For illustration purposes, we will often consider a \$-padded extension of string t, denoted $t^+ := t\$^n$. For $1 \le i \le n$, let $s_i(t) = t_i \dots t_n$ indicate the i^{th} (non-empty) suffix of string t. The starting position i is called its *suffix number*.

The *suffix array* $sa(t)$ of t is a permutation of the suffix numbers $\{1 \dots n\}$, according to the lexicographic ordering of the n suffixes of t. More precisely, for all pairs of indices (k, l), $1 \le k < l \le n$, the suffix $s_{sa[k]}(t)$ at position k in the suffix array is lexicographically smaller than the suffix $s_{sa[l]}(t)$ at position l in the suffix array.

A bucket $b = [l, r]$ is an interval of the suffix array sa, determined by its left and right end in sa. A bucket $b_p = [l, r]$ is called a level-m bucket, if all contained suffixes $s_{sa[l]}(t), s_{sa[l+1]}(t), \dots, s_{sa[r]}(t)$ share a common prefix p of length m.

A *radix step* denotes the part of an algorithm in which strings are sorted according to the characters at a certain *offset* in the string. The *offset* is called *radix level*. A radix step is like a single iteration of *radix sort*.

Range reduction performs a bijective function, $rank : \Sigma \to \{0, \dots, |\Sigma| - 1\}$, of the alphabet to the first $|\Sigma|$ natural numbers, while keeping the alphabetical order. More precisely, for two characters $c_1 < c_2$ of alphabet Σ, $rank(c_1)$ is smaller than $rank(c_2)$. Applied to a string t, *range reduction* maps each character of t according to its *rank*. Note that the suffix array of a range reduced string equals the suffix array of the original string.

A *multiple character encoding* for strings of length d is a bijective function $code_d : \Sigma^d \to \{0, \dots, |\Sigma|^d - 1\}$ such that for strings u, v of length d, $code_d(u) < code_d(v)$ if and only if u is lexicographically smaller than v.

For a given rank function as defined above, such an encoding can easily be defined as $code_d(u) = \sum_{k=1}^{d} |\Sigma|^{d-k} \cdot rank(u[i + k - 1])$. The encoding can be generalized to strings of length greater than d, by just encoding the first d characters. Given the encoding $code_d(i)$ for suffix $s_i(t)$, $1 \le i < n$, the encoding for suffix $s_{i+1}(t)$ can be derived by shifting away the first character of $s_i(t)$ and adding the *rank* of character $t^+[i + d]$:

$$code_d(i+1) = |\Sigma| \left(code_d(i) \bmod |\Sigma|^{d-1} \right) + rank(t^+[i+d]). \quad (2.1)$$

3 The Bucket-Pointer Refinement algorithm.

Most of the previously mentioned practical algorithms perform an ordering of suffixes concerning leading characters into buckets, followed by a refinement of these buckets, often recursively. Before describing our algorithm that uses a similar though somewhat enhanced technique, we classify the specific techniques used.

3.1 Classification of techniques. The first type of bucket refinement techniques found in the literature is formed by string sorting methods without using the dependencies among suffixes. Most representatives of this class sort the suffixes concerning leading characters and then refine the groups of suffixes with equal prefix by recursively performing radix steps until unique prefixes are obtained [1, 2, 19].

String to build suffix array for: $t =$ e f e f e f a a (positions 1 2 3 4 5 6 7 8)

	a$	aa		ef		fa	fe	
sa after initial sorting ($d = 2$):	8	7	1	3	5	6	2	4
	1	2	3	4	5	6	7	8
bptr after initial sorting:	5	8	**5**	8	**5**	6	**2**	1
sa after sorting bucket [3,5]:	8	7	5	1	3	6	2	4
	1	2	3	4	5	6	7	8
bptr after updating positions 1, 3, 5:	**5**	8	**5**	8	**3**	6	2	1

Figure 1: Example of the refinement procedure after the initial sorting of suffixes concerning prefixes of length $d = 2$. The suffix array *sa* and the table of bucket pointers *bptr* is shown before and after applying the refinement steps to the bucket $[3, 5]$ containing suffixes 1, 3, 5 with respect to *offset* $= 2$. The sort keys (drawn in bold face) are $sortkey(1) = bptr[1 + 2] = 5$, $sortkey(3) = bptr[3 + 2] = 5$, and $sortkey(3) = bptr[5 + 2] = 2$. After the sorting concerning the bucket pointers, the bucket pointers for the suffixes 1, 3, 5 in bucket $[3, 5]$ are updated to $bptr[1] = 5$, $bptr[3] = 5$, and $bptr[5] = 3$.

The second type of algorithms use the order of already computed suffixes in the refinement phases. If suffixes $s_i(t)$ and $s_j(t)$ share a common prefix of length *offset*, the order of $s_i(t)$ and $s_j(t)$ can be derived from the ordering of suffixes $s_{i+offset}(t)$ and $s_{j+offset}(t)$. Many practical algorithms using this techniques also apply methods of the first type as a fall-back, when the order of suffixes at *offset* is not yet available [9, 18, 20].

We further divide the second type in two subtypes, the *push* and *pull* methods. The *push* method uses the ordering of already determined groups sharing a leading character to forward its ordering to undetermined buckets. Representatives are Itoh and Tanaka's *two-stage* algorithm [9], Seward's *copy* algorithm [20], and the *deep-shallow* algorithm of Manzini and Ferragina [18] that uses this technique among some others.

Pull methods look up the order of suffixes $s_{i+offset}(t)$ and $s_{j+offset}(t)$ to determine the order of $s_i(t)$ and $s_j(t)$. This technique is used in many algorithms. Representatives among practical algorithms are Larsson and Sadakane's *qsufsort* [16], Seward's *cache* algorithm [20], and *deep-shallow* sorting of Manzini and Ferragina [18]. The *difference-cover* algorithm by Burkhardt and Kärkkäinen [3] first sorts a certain subset of suffixes to ensure the existence of a bounded offset to these already ordered suffixes in the final step.

3.2 The new algorithm. Now, we will describe our new *bucket-pointer refinement* (*bpr*) algorithm. The algorithm melds the approaches of refining groups with equal prefix by recursively performing radix steps and the *pull* technique.

It mainly consists of two simple phases. Given a parameter d (usually less than $\log n$), in the first phase the suffixes are lexicographically sorted, such that afterwards suffixes with same d-length prefix are grouped together. Before entering the second phase, for each suffix with suffix number i a pointer to its bucket $bptr[i]$ is computed, such that suffixes with same d-length prefix share the same bucket pointer. Lexicographically smaller suffixes have smaller bucket pointer. The position of the rightmost suffix in each bucket can be used as bucket pointer.

In the second phase, the buckets containing suffixes with equal prefix are recursively refined. Let $[l, r]$ be the segment in the suffix array *sa* forming a bucket B of suffixes $sa[l], sa[l + 1], \ldots, sa[r]$ with equal prefix of length *offset*. Then, the refinement procedure works in the following way. The suffixes in B are sorted according to the bucket pointer at *offset*. That is, for each suffix $sa[k]$ in B, $l \leq k \leq r$, $bptr[sa[k] + offset]$ is used as sort key.

After sorting the suffixes of B, the sub-buckets each containing suffixes sharing the same sort key are determined, and for each suffix $sa[k]$, $l \leq k \leq r$, the bucket pointer $bptr[sa[k]]$ is updated to point to the new sub-bucket containing $sa[k]$. Finally, all sub-buckets are refined recursively by calling the refinement procedure with increased offset, $offset_{new} = offset_{old} + d$. After termination of the algorithm, *sa* is the suffix array and the array *bptr* reflects the inverse suffix array. An example of the refinement procedure is given in Figure 1.

Properties. The main improvement of our algorithm, compared to earlier algorithms performing bucket refinements, is that it benefits from the immediate use of subdivided bucket pointers after each re-

finement step. With increasing number of subdivided buckets, it becomes more and more likely that different bucket pointers can be used as sort keys during a refinement step, such that the expected recursion depth decreases for the buckets refined later. The final position of a suffix i in the current bucket is reached at the latest when $bptr[i + offset]$ is unique for the current *offset*, that is, when the suffix $i+offset$ has already reached its final position.

Another improvement of our algorithm is that in each recursion step the *offset* can be increased by d. Hence, the recursion depth decreases by a factor of d, compared to algorithms performing characterwise radix steps.

Note that the algorithm can be applied to arbitrary ordered alphabets, since it just uses comparisons to perform suffix sorting.

Analysis. We were so far not able to determine tight time bounds for our algorithm. The problem is that the algorithm quite arbitrarily uses the dependence among suffixes. Hence, we can only state a straight forward quadratic time complexity for the general worst case, while a subquadratic upper time bound can be found for certain periodic strings.

The simple $O(n^2)$ upper time bound can be seen as follows. The first phase of the algorithm can simply be performed in linear time (see Section 3.3 for more details). For the second phase, we assume to apply a sorting procedure in $O(n \log n)$ time. On each recursion level there are at most n suffixes to be sorted in $O(n \log n)$ time. The maximal *offset* until the end of the string is reached is n, and the *offset* is incremented by d in each recursive call. Hence, the maximal recursion depth is $\frac{n}{d}$. Therefore, the worst-case time complexity of the algorithm is limited by $O\left(\frac{n^2 \log n}{d}\right)$. By setting $d = \log n$, we obtain an upper bound of $O(n^2)$ without taking into account the dependencies among suffixes.

Now, we focus on especially bad instances for our algorithm, in particular strings maximizing the recursion depth. Since the recursion depth is limited by the LCPs of suffixes to be sorted, periodic strings maximizing the average LCP are especially hard strings for our algorithm.

A string $\mathtt{a}^n$ consisting of one repeated character maximizes the average LCP and is therefore analyzed as a representative for bad input strings. In the first phase of our algorithm the last $d-1$ suffixes $s_{n+2-d}(\mathtt{a}^n), s_{n+3-d}(\mathtt{a}^n), \ldots, s_n(\mathtt{a}^n)$ are mapped to singleton buckets. One large bucket containing all the other suffixes with leading prefix $\mathtt{a}^d$ remains to be refined. In each recursive refinement step, the remaining large bucket is again subdivided into *offset* singleton buckets and one larger bucket. In each recursive refinement step the *offset* is incremented by d, starting with $offset = d$ in step 1. Hence, in the i-th refinement step $i \cdot d$ suffixes are subdivided into singleton buckets. The recursion proceeds until all buckets are singleton, that is, until a recursion depth *recdepth* is reached, such that $n \leq d - 1 + \sum_{i=1}^{recdepth} i \cdot d = d - 1 + d \cdot \frac{recdepth(recdepth+1)}{2}$. Therefore, for the string $\mathtt{a}^n$ the recursion depth *recdepth* of our algorithm is $\Theta\left(\sqrt{\frac{n}{d}}\right)$.

Less than n suffixes have to be sorted in each recursive step. Hence, we multiply sorting complexity and recursion depth *recdepth* to get the time bound $O\left(n \log n \sqrt{\frac{n}{d}}\right)$ of our algorithm for the string $\mathtt{a}^n$. By setting $d = \log n$ we achieve a running time of $O\left(n \log n \sqrt{\frac{n}{\log n}}\right) = O\left(n^{\frac{3}{2}} \sqrt{\log n}\right)$. By taking into account the decreasing number of suffixes to be sorted with increasing recursion depth, more sophisticated analysis shows the same time bound. Therefore, this worst-case time bound seems to be tight for this string.

3.3 Engineering and Implementation. Now, we give a more detailed description of the phases of the algorithm and briefly explain enhancements of the basic algorithm to achieve faster construction times.

Phase 1. We perform the initial sorting concerning the d-length prefixes of the suffixes by bucket sort, using $code_d(i)$ as the sort key for suffix i, $1 \leq i \leq n$.

The bucket sorting is done by two scans of the sequence, thereby successively computing $code_d(i)$ for each suffix using equation (2.1). There are $|\Sigma|^d$ buckets, one for each possible $code_d$. In the first scan, the size of each bucket is determined by counting the number of suffixes for each possible $code_d$. The outcome of this is used to compute the starting position for each bucket, stored in table *bkt* of size $|\Sigma|^d$. During the second scan, the suffix numbers are mapped to the buckets, where suffix number i is mapped to bucket number $code_d(i)$.

After the bucket sort, the computation of the bucket pointer table *bptr* can be done by another scan of the sequence. For suffix i, the bucket pointer $bptr[i]$ is simply the rightmost position of the bucket containing i, $bptr[i] = bkt[code_d(i) + 1] - 1$.

Phase 2. We now give a more detailed description of the three steps of the refinement procedure and present improvements to the basic approach.

Sorting. In the refinement procedure, first the suffixes are sorted with respect to a certain *offset* using the bucket pointer $bptr[i + offset]$ as the sort key for suffix i. The used sorting algorithms are well known. *Insertion Sort* is used for buckets of size up to 15. For the larger buckets, we apply *Quicksort* using the

partitioning scheme due to Lomuto [5, Problem 8-2]. The pivot is chosen to be the median of 3 elements due to Singleton [21]. Further on, we apply a heuristic for the case that we have many equal bucket pointers: After a partitioning step, we just extend the partitioning position as long as the sort key equals the pivot, such that there is an already sorted region around this position and the size of the remaining unsorted partitions decreases. Especially for periodic strings, this heuristic works very well.

Updating bucket pointers. The update of bucket pointers can simply be performed by a right-to-left scan of the current bucket. As long as the sort keys of consecutive suffixes are equal, they are located in the same refined bucket, and the bucket pointer is set to the rightmost position of the refined bucket. Note that the refined bucket positions are implicitly contained in the bucket pointer table *bptr*. The left pointer l of a bucket is the right pointer of the bucket directly to the left increased by one, and the right pointer r is simply the bucket pointer for the suffix $sa[l]$ at position l, $r = bptr[sa[l]]$, since for each suffix i its bucket pointer $bptr[i]$ points to the rightmost position of its bucket.

Recursive Refinement. The recursive refinement procedure is usually called with incremented offset, *offset*+d. Note that for a sub-bucket b_{sub} of b containing each suffix $s_i(t)$ for which also suffix $s_{i+offset}(t)$ with respect to *offset* is contained in b, the new *offset* can be doubled. This is so because all suffixes contained in b share a common prefix of length *offset*, and for each suffix $s_i(t)$ in b_{sub} also the suffix $s_{i+offset}(t)$ with respect to *offset* is in b. Hence, all suffixes contained in b_{sub} share a prefix of $2 \cdot$ *offset*.

Further on, we add an additional heuristic to avoid the repeated useless sorting of buckets. If we meet a bucket consisting of suffixes all sharing a common prefix much larger than the current *offset*, we might perform many refinement steps, without actually refining the bucket, until the *offset* reaches the length of the common prefix. Therefore, if a bucket is not refined during a recursion step, we search for the lowest *offset* dividing the bucket. This is performed by just iteratively scanning the bucket pointers of the contained suffixes with respect to *offset* and incrementing the *offset* by d after each run. As soon as a bucket pointer different from the others is met, the current *offset* is used to call the refinement procedure.

Further improvements. We enhanced our algorithm with the *copy* method by Seward [20] that was earlier mentioned by Burrows and Wheeler [4]. If the buckets consisting of suffixes with leading character p are determined, they form a level-1 bucket B_p. Let $b_{c_1p}, b_{c_2p}, \ldots b_{c_{|\Sigma|}p}$, $c_i \in \Sigma$, be the level-2 buckets with second character p. The ordering of suffixes in B_p can be used to forward the ordering to the specified level-2 buckets by performing a single path over B_p. If i is the current suffix number in B_p and $c = t[i-1]$ is the previous character, then the suffix $i-1$ is written to the first non-determined position of bucket b_{cp}. Seward also showed how to derive the positions of suffixes in b_{pp} using the buckets b_{c_ip}, $p \neq c_i \in \Sigma$. Hence, the usage of Seward's *copy* technique avoids the comparison based sorting of more than half of the buckets.

Our program processes the level-1 buckets B_c, $c \in \Sigma$, in ascending order with respect to the number of suffixes, $|B_c| - |b_{cc}|$, to be sorted comparison-based.

4 Experimental Results.

In this section we investigate the practical construction times of our algorithm for DNA sequences, common texts, and artificially generated strings with high average LCP.

We compared the *bpr* implementation (available through `http://bibiserv.techfak.uni-bielefeld.de/bpr/`) to seven other practical implementations: *deep-shallow* by Manzini and Ferragina [18], *cache* and *copy* by Seward [20], *qsufsort* by Larsson and Sadakane [16], *difference-cover* by Burkhardt and Kärkkäinen [3], *divide-and-conquer* by Kim *et al.* [12], and *skew* by Kärkkäinen and Sanders [10]. Since the original *skew* implementation is working on integer alphabets, in all instances we mapped the character string to an integer array.

The experiments were performed on a computer with 512 MBytes of main memory and *1.3 GHz Intel Pentium^{TM}M* processor, running the Linux operating system. All programs were compiled with the *gcc*-compiler, respectively *g++*-compiler, with optimization options '-O3 -fomit-frame-pointer -funroll-loops'.

The investigated data files are listed in Table 1, ordered by average LCP. For the DNA sequences, we selected genomic DNA from different species: The whole genome of the bacteria *Escherichia coli* (*E. coli*), the fourth chromosome of the flowering plant *Arabidopsis thaliana* (*A. thaliana*), the first chromosome of the nematode *Caenorhabditis elegans* (*C. elegans*), and the human chromosome 22. Additionally, we investigated the construction times for different concatenated DNA sequences of certain families. We used six *Streptococcus* genomes, four genomes of the *Chlamydophila* family, and three different *E. coli* genomes.

For the evaluation of other common real-world strings, we used the suite of test files given by Manzini and Ferragina in [18]. The strings are usually concatenations of text files, respectively *tar*-archives. Due to memory constraints on our test computer, we just took

data set	LCP		string	alphabet	description
	average	maximum	length	size	
E. coli genome	17	2815	4,638,690	4	*Escherichia coli* genome
A. thaliana chr. 4	58	30,319	12,061,490	7	*A. thaliana* chromosome 4
Human chr. 22	1979	199,999	34,553,758	5	*H. sapiens* chromosome 22
C. elegans chr. 1	3,181	110,283	14,188,020	5	*C. elegans* chromosome 1
6 *Streptococci*	131	8,091	11,635,882	5	6 *Streptococcus* genomes
4 *Chlamydophila*	1,555	23,625	4,856,123	6	4 *Chlamydophila* genomes
3 *E. coli*	68,061	1,316,097	14,776,363	5	3 *E. coli* genomes
bible	13	551	4,047,392	63	King James bible of the Canterbury Corpus
world192	23	559	2,473,400	94	CIA world fact book of Canterbury Corpus
rfc	87	3,445	50,000,000	110	Concatenated texts from the RFC project
sprot34	91	2,665	50,000,000	66	SwissProt database
howto	267	70,720	39,422,105	197	Concatenation of Linux Howto files
reuters	280	24,449	50,000,000	91	Reuters news in XML
w3c	478	29,752	50,000,000	255	Concatenated html files from w3c homepage
jdk13	654	34,557	50,000,000	110	Concatenation of JDK 1.3 doc files
linux	766	136,035	50,000,000	256	tar-archive of the linux kernel source files
etext99	1,845	286,352	50,000,000	120	Concatenated texts from Project Gutenberg
gcc	14,745	856,970	50,000,000	121	tar-archive of *gcc* 3.0 source files
random	4	9	20,000,000	26	Bernoulli string
period 500,000	9,506,251	19,500,000	20,000,000	26	Repeated Bernoulli string
period 1,000	9,999,001	19,999,000	20,000,000	26	Repeated Bernoulli string
period 20	9,999,981	19,999,980	20,000,000	17	Repeated Bernoulli string
Fibonacci	5,029,840	10,772,535	20,000,000	2	*Fibonacci string*

Table 1: Description of the data set.

the last 50 million characters of those text files that exceed the 50 million character limit.

The artificial files have been generated as described by Burkhardt and Kärkkäinen [3]: a random string made out of Bernoulli distributed characters and periodic strings composed of an initial random string, repeated until a length of 20 million characters is reached. We have used initial random strings of length 20, 1,000 and 500,000 to generate the periodic strings. Also, we investigated a Fibonacci string of length 20 million characters.

The suffix array construction times of the different algorithms are given in Tables 2–4. Table 2 contains the construction times for the DNA sequences. Our *bpr* algorithm is the fastest suffix array construction algorithm for most DNA sequences. Only *deep-shallow* is about 5% faster for the fourth chromosome of *A. thaliana*. But for sequences with higher average LCP, *bpr* outperforms all algorithms. For the concatenated sequence of 3 *E. coli* genomes with average LCP 68,061, *deep-shallow*, the closest competitor of *bpr*, is 1.72 times slower.

For common real-world strings the running times of the investigated algorithms are shown in Table 3. For the texts with small average LCP, *deep-shallow* is the fastest suffix array construction algorithm. *Bpr* shows the second best running times. But when the average LCP exceeds a limit of about 300, our algorithm is the fastest among all investigated algorithms. For *gcc* with average LCP 14,745 it is by far the fastest suffix array construction algorithm. *Deep-shallow* is 2.30 times slower (76.23 vs. 33.19 seconds). The second fastest algorithm for *gcc* is *qsufsort*. It requires 59.44 seconds and therefore is 1.79 times slower than *bpr*.

For degenerated strings the construction times are given in Table 4. We have stopped when an algorithm used more than 12 hours of computation time, indicated by a dash in Table 4. For the degenerated strings, *bpr* is tremendously faster than *deep-shallow*, *cache*, and *copy*, which are very fast algorithms for strings with lower average LCP. Even compared to the algorithms *qsufsort*, *difference-cover*, *divide-and-conquer*, and *skew* with good worst-case time complexity our algorithm performs very well. For strings with period 1,000 and 500,000, it is by far the fastest among all tested suffix array construction algorithms. For strings with

DNA sequences	construction time (sec.)							
	bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide conquer*	*skew*
E. coli genome	**1.46**	1.71	3.69	2.89	2.87	4.32	5.81	13.48
A. thaliana chr. 4	5.24	**5.01**	12.24	9.94	8.42	13.29	16.94	38.30
Human chr. 22	**15.92**	16.64	40.08	30.04	26.52	44.93	51.31	112.38
C. elegans chr. 1	**5.70**	6.03	20.79	17.48	13.09	16.94	18.64	41.28
6 *Streptococci*	**5.27**	5.97	14.43	10.38	13.16	14.50	16.40	36.24
4 *Chlamydophila*	**2.31**	3.43	17.46	14.45	7.49	5.59	6.13	14.13
3 *E. coli*	**8.01**	13.75	437.18	1,294.30	32.72	20.57	21.58	47.32

Table 2: Suffix array construction times for different DNA sequences and generalized DNA sequences by different algorithms, with $d = 7$ for bpr.

text	construction time (sec.)							
	bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide conquer*	*skew*
bible	1.90	**1.41**	2.91	2.24	3.17	3.74	6.39	11.59
world192	1.05	**0.73**	1.47	1.24	1.91	2.28	3.57	6.45
rfc	31.16	**26.37**	57.97	55.21	58.10	71.10	101.57	169.03
sprot34	35.75	**29.77**	71.95	71.96	60.24	81.76	104.71	169.16
howto	22.10	**19.63**	39.92	47.27	41.14	48.45	83.32	141.50
reuters	**47.32**	52.74	111.80	157.63	73.19	108.85	108.84	169.18
w3c2	**41.04**	61.37	82.46	167.76	69.40	96.02	105.89	163.15
jdk13	**40.35**	47.23	101.58	263.86	73.75	97.12	98.13	162.39
linux	**23.72**	23.95	50.93	99.47	61.01	65.66	98.06	173.05
etext99	**32.60**	33.25	68.84	267.48	61.19	65.31	110.95	190.33
gcc	**33.19**	76.23	2,894.81	21,836.56	59.44	73.54	83.96	162.06

Table 3: Suffix array construction times for different text by different algorithms, with $d = 3$ for bpr.

artificial strings	construction time (sec.)							
	bpr	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*divide conquer*	*skew*
random	8.95	**8.31**	15.17	10.83	14.83	20.19	37.52	47.25
period 500,000	**12.33**	710.52	–	–	89.52	47.28	31.21	61.04
period 1,000	**16.76**	1,040.45	–	–	86.45	78.87	21.96	52.34
period 20	41.61	–	–	–	74.74	59.38	**10.33**	43.24
Fibonacci string	28.00	680.43	–	–	82.62	69.63	**22.21**	38.17

Table 4: Suffix array construction times for artificial strings by different algorithms, with $d = 3$ for bpr.

period 20 and for the Fibonacci string, just the *divide-and-conquer* algorithm with its $O(n \log \log n)$ worst-case time complexity is faster.

In summary, one can say that *bpr* is always among the two fastest of the investigated algorithms. In most cases, especially all but one DNA sequences, it is even the fastest one. For strings with very small average LCP its running time is comparable to *deep-shallow*, *cache*, and *copy*. With growing average LCP, *bpr* is clearly the fastest algorithm. Even for worst-case strings with very high average LCP, *bpr* performs well compared to algorithms *qsufsort*, *difference-cover*, and *divide-and-conquer* with good worst-case time complexity, whereas the construction times for *deep-shallow*, *cache*, and *copy* escalate.

We believe that the practical speed of our algorithm is mainly due to the combination of different techniques with good locality behavior and the employment of relations among suffixes. The bucket sort in phase 1 has optimal locality of memory access regarding the input string that is just scanned twice. If the d-length substrings are uniformly distributed, phase 1 equally divides all suffixes into small buckets. In phase 2 each bucket is refined recursively until it consists of singleton sub-buckets. This technique of dividing suffixes from small to smaller buckets is similar to *Quicksort* for original sorting. It has good locality of memory access. Accordingly, it is fast in practice. Further on, by using the bucket pointers as sort keys, our method incorporates information about subdivided buckets into the bucket refinement process as soon as this information becomes available. The combination of these techniques, further heuristics in the refinement procedure (Section 3.3), and the *copy* method [20] result in the final fast practical algorithm.

However, the space requirements of *bpr* are higher than the space requirements for *deep-shallow*, *cache*, and *copy*. It takes $8n$ bytes for the suffix array and the bucket pointer table. Additional space is used for the bucket pointers of the initial bucket sort and for the recursion stack, though the recursion depth decreases by a factor of d. However, for certain applications, e.g. the computation of the Burrows-Wheeler-Transform [4], the construction of the suffix array is just a byproduct, such that not the complete suffix array needs to remain in memory.

Therefore, if one is concerned about space, the *deep-shallow* algorithm might be the best choice. If there are no major space limitations, we believe that the *bpr* algorithm is the best practical choice.

5 Conclusion and further work.

We have presented a fast suffix array construction algorithm that performs very well even for worst-case strings. Due to its simple structure, it is easy to implement. Therefore, we believe that our algorithm can be widely used in all kinds of suffix array applications.

An open question remains. We were so far unable to prove a better worst-case time complexity than $O(n^2)$. For certain periodic strings, we verified an $O\left(n^{\frac{3}{2}}\sqrt{\log n}\right)$ time bound, but for general strings finding a non-trivial upper bound seems to be hard since our algorithm quite arbitrarily uses the dependence among suffixes.

Of further interest will be the parallelization of suffix array construction, since the suffix array construction for very large DNA sequences is usually performed on servers with more than one CPU.

Acknowledgments. We thank Dong Kyue Kim for providing the code of the *divide-and-conquer* algorithm, Peter Husemann for implementing the timing for the *skew* algorithm, and Hans-Michael Kaltenbach for advice on the analysis of the *bpr* algorithm.

References

[1] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, November 1993.

[2] J. L. Bentley and R Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1997)*, pages 360–369, January 1997.

[3] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer Verlag, June 2003.

[4] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report Research Report 124, Digital System Research Center, May 1994.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, first edition, 1989.

[6] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 1997)*, pages 137–143, October 1997.

[7] M. Farach, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, November 2000.

[8] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices.

In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003)*, pages 251–260. IEEE Computer Society, October 2003.

[9] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of String Processing and Information Retrieval Symposium and International Workshop on Groupware, (SPIRE/CRIWG 1999)*, pages 81–88. IEEE Computer Society Press, September 1999.

[10] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer Verlag, June 2003.

[11] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th ACM Symposium on Theory of Computing (STOC 1972)*, pages 125–136. ACM Press, May 1972.

[12] D. K. Kim, J. Jo, and H. Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In Celso C. Ribeiro and Simone L. Martins, editors, *Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms (WEA 2004)*, volume 3059 of *Lecture Notes in Computer Science*, pages 301–314. Springer Verlag, May 2004.

[13] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer Verlag, June 2003.

[14] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer Verlag, June 2003.

[15] S. Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, November 1999.

[16] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, May 1999.

[17] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[18] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, June 2004.

[19] P.M. McIlroy, K. Bostic, and M.D. McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[20] J. Seward. On the performance of BWT sorting algorithms. In *Proceedings of the Data Compression Conference (DCC 2000)*, pages 173–182. IEEE Computer Society, March 2000.

[21] R. C. Singleton. ACM Algorithm 347: an efficient algorithm for sorting with minimal storage. *Communications of the ACM*, 12(3):185–187, March 1969.